ITU FRP 2010

Lecture 2: Yampa: Arrows-based FRP

Henrik Nilsson

School of Computer Science

University of Nottingham, UK

Outline

- CFRP issues
- Introduction to Yampa
- Arrows
- A closer look at Yampa

CFRP issues: Sharing

Consider:

```
let x = 1 + integral (x * x) in x
```

The recursively defined behavior, a ***function***, is applied over and over to the ***same*** stream of sample times.

- Causes recomputation
- Laziness does ***not*** help
- Memoization needed to get acceptable performance. But with care to avoid memory leaks.

CFRP issues: Restart (1)

Consider:

```
let
    c = hold 0 (count (repeatedly 0.5))
in
    c ‘until‘ after 5 -=> c * 2
```

What happened at the time of the switch?

- CFRP behaviors and events are ***signal generators***: they will start from scratch when swicthched in.
- But what if we just want to continue observing an evolving signal?

CFRP issues: Restart (2)

- A version of `until` that starts new behaviors from time 0.
  Time and space leak!
- Support signals as well, e.g. through some variant of `runningIn`:

  ```
  runningIn ::
          B a -> (B a -> B b) -> B b
  ```

  Idea: apply behavior to start time once and for all, then wrap up the resulting signal as a signal generator that ignores its starting time.

CFRP issues: Restart (3)

Problems with `runningIn`

- No type-level distinction between signals and signal generators: a “running behavior” is a signal masquerading as a signal generator. (But could be fixed though other designs.)
- Difficult to implement; requires imperative techniques, implies certain overhead.

An alternative

By adopting ***signal functions*** as the central notion, these problems are side stepped:

- Sharing amounts to sharing computations of signal samples: lazy evaluation handles that just fine.
- Observation of externally originating signals is inherent in the notion of a signal function.
- Implementation is straightforward.

Yampa

What is ***Yampa***?

- FRP implementation structured using ***arrows***.

Yampa

What is ***Yampa***?

- FRP implementation structured using ***arrows***.
- Realised as an ***Embedded Domain-Specific Language*** (EDSL), i.e. a combinator library.

Yampa

What is ***Yampa***?

- FRP implementation structured using ***arrows***.
- Realised as an ***Embedded Domain-Specific Language*** (EDSL), i.e. a combinator library.
- ***Continuous-time*** signals (conceptually)

Yampa

What is ***Yampa***?

- FRP implementation structured using ***arrows***.
- Realised as an ***Embedded Domain-Specific Language*** (EDSL), i.e. a combinator library.
- ***Continuous-time*** signals (conceptually)
- Discrete-time signals represented by continuous-time signal carrying option type ***`Event`***.

Yampa

What is ***Yampa***?

- FRP implementation structured using ***arrows***.
- Realised as an ***Embedded Domain-Specific Language*** (EDSL), i.e. a combinator library.
- ***Continuous-time*** signals (conceptually)
- Discrete-time signals represented by continuous-time signal carrying option type ***Event***.
- Functions on signals, ***Signal Functions***, is the central abstraction, forming the arrows.

Yampa

- Signal functions are first-class entities, signals a secondary notion, only existing indirectly through the signal functions.

Yampa

- Signal functions are first-class entities, signals a secondary notion, only existing indirectly through the signal functions.
- Advanced ***switching constructs*** to describe systems with highly dynamic structure.

Yampa

- Signal functions are first-class entities, signals a secondary notion, only existing indirectly through the signal functions.
- Advanced ***switching constructs*** to describe systems with highly dynamic structure.
- People:
  - Antony Courtney
  - Paul Hudak
  - Henrik Nilsson
  - John Peterson

Yampa?

Yampa?

Yet

Another

Mostly

Pointless

Acronym

Yampa?

Yet

Another

Mostly

Pointless

Acronym

???

Yampa?

Yet

Another

Mostly

Pointless

Acronym

???

No . . .

Yampa?

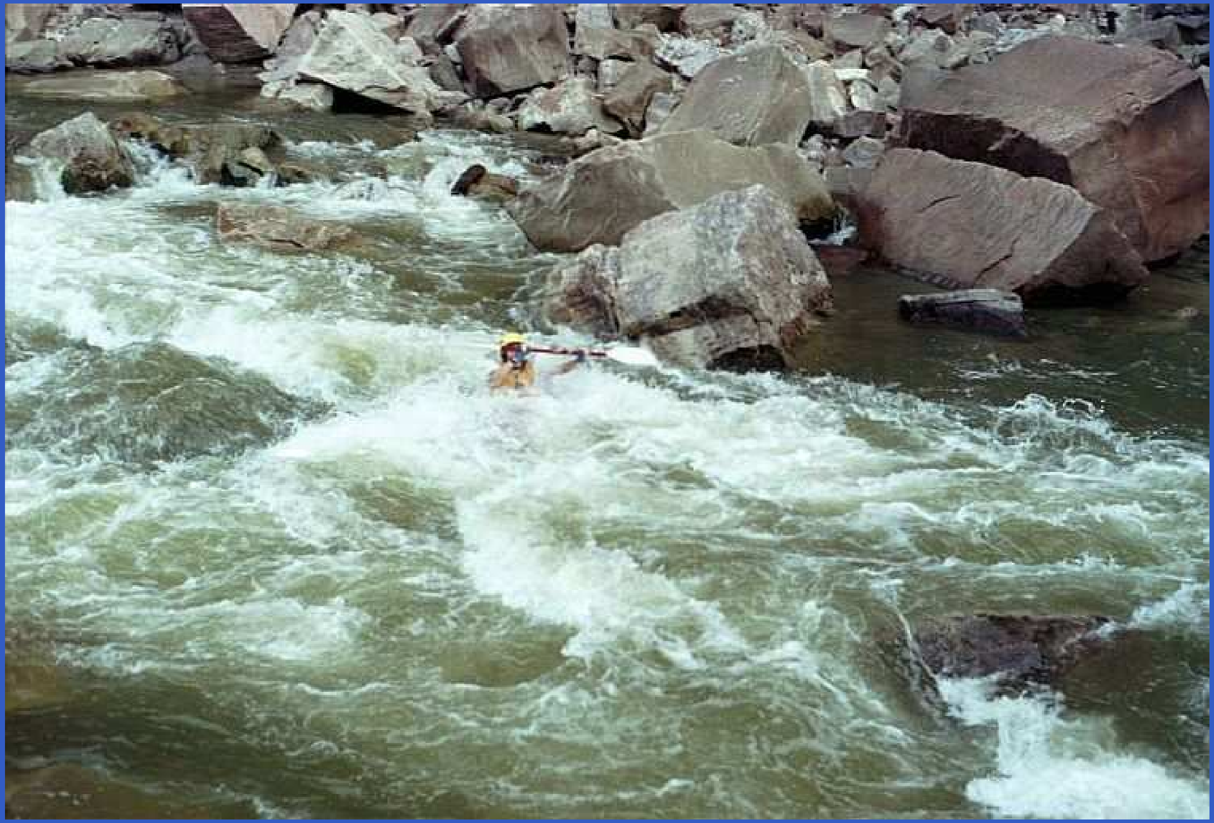
Yampa is a river . . .

Yampa?

. . . with long calmly flowing sections . . .

Yampa?

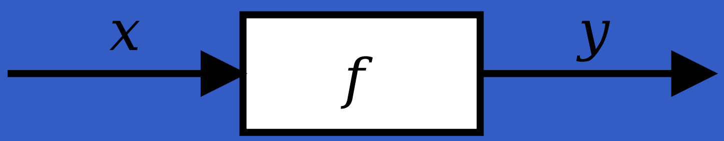
. . . and abrupt whitewater transitions in between.



A good metaphor for hybrid systems!

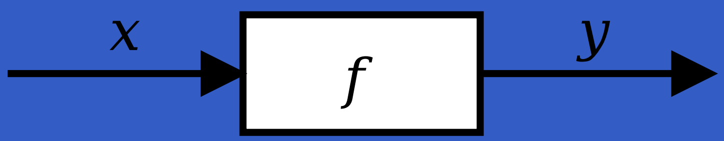
Signal functions (1)

Key concept: ***functions on signals***.



Signal functions (1)

Key concept: ***functions on signals***.



Intuition:

$$\texttt{Signal}\ \alpha \approx \texttt{Time} \rightarrow \alpha$$
$$\texttt{SF}\ \alpha\ \beta \approx \texttt{Signal}\ \alpha \rightarrow \texttt{Signal}\ \beta$$

x `:: Signal T1`
y `:: Signal T2`
f `:: SF T1 T2`

Signal functions (2)

Additionally, ***causality*** required: output at time t must be determined by input on interval $[0, t]$.

Signal functions (2)

Additionally, ***causality*** required: output at time t must be determined by input on interval $[0, t]$.

Signal functions are said to be

- ***pure*** or ***stateless*** if output at time t only depends on input at time t

Signal functions (2)

Additionally, ***causality*** required: output at time t must be determined by input on interval $[0, t]$.

Signal functions are said to be

- ***pure*** or ***stateless*** if output at time t only depends on input at time t
- ***impure*** or ***stateful*** if output at time t depends on input over the interval $[0, t]$.

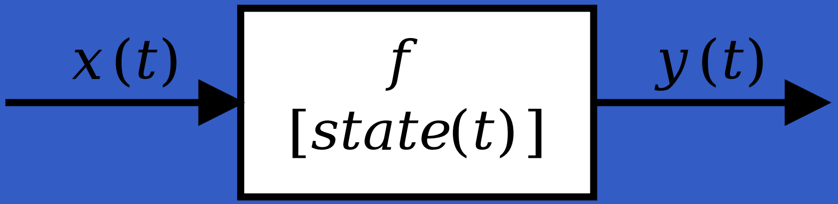
Signal functions and state

Alternative view:

Signal functions and state

Alternative view:

Signal functions can encapsulate ***state***.

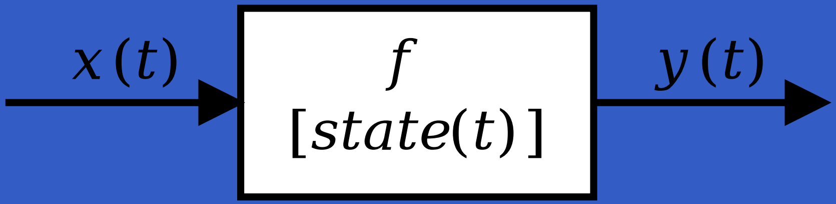


$state(t)$ summarizes input history $x(t')$, $t' \in [0, t]$.
Thus, really a kind of ***process***.

Signal functions and state

Alternative view:

Signal functions can encapsulate ***state***.



$state(t)$ summarizes input history $x(t')$, $t' \in [0, t]$.
Thus, really a kind of ***process***.

From this perspective, signal functions are:

- ***stateful*** if $y(t)$ depends on $x(t)$ and $state(t)$
- ***stateless*** if $y(t)$ depends only on $x(t)$

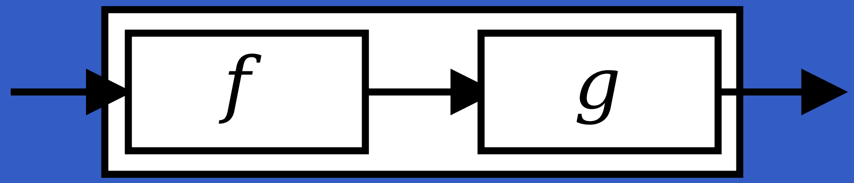
Yampa and arrows (1)

In Yampa, systems are described by combining signal functions (forming new signal functions).

Yampa and arrows (1)

In Yampa, systems are described by combining signal functions (forming new signal functions).

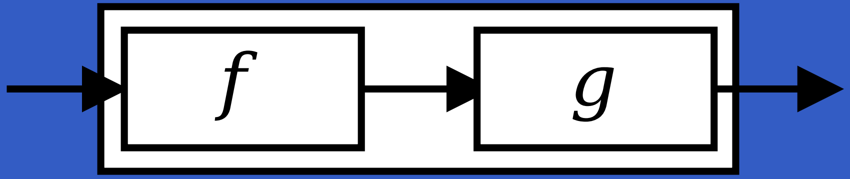
For example, serial composition:



Yampa and arrows (1)

In Yampa, systems are described by combining signal functions (forming new signal functions).

For example, serial composition:

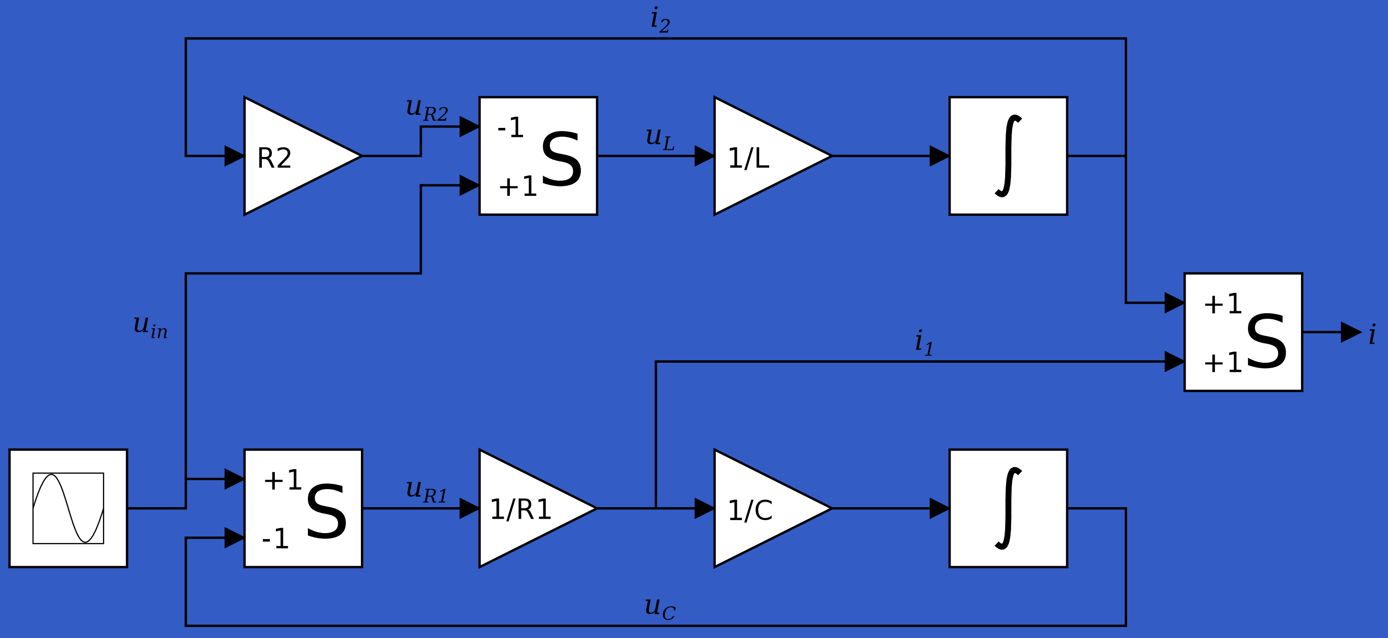


A *combinator* can be defined that captures this idea:

```
(>>>) :: SF a b -> SF b c -> SF a c
```

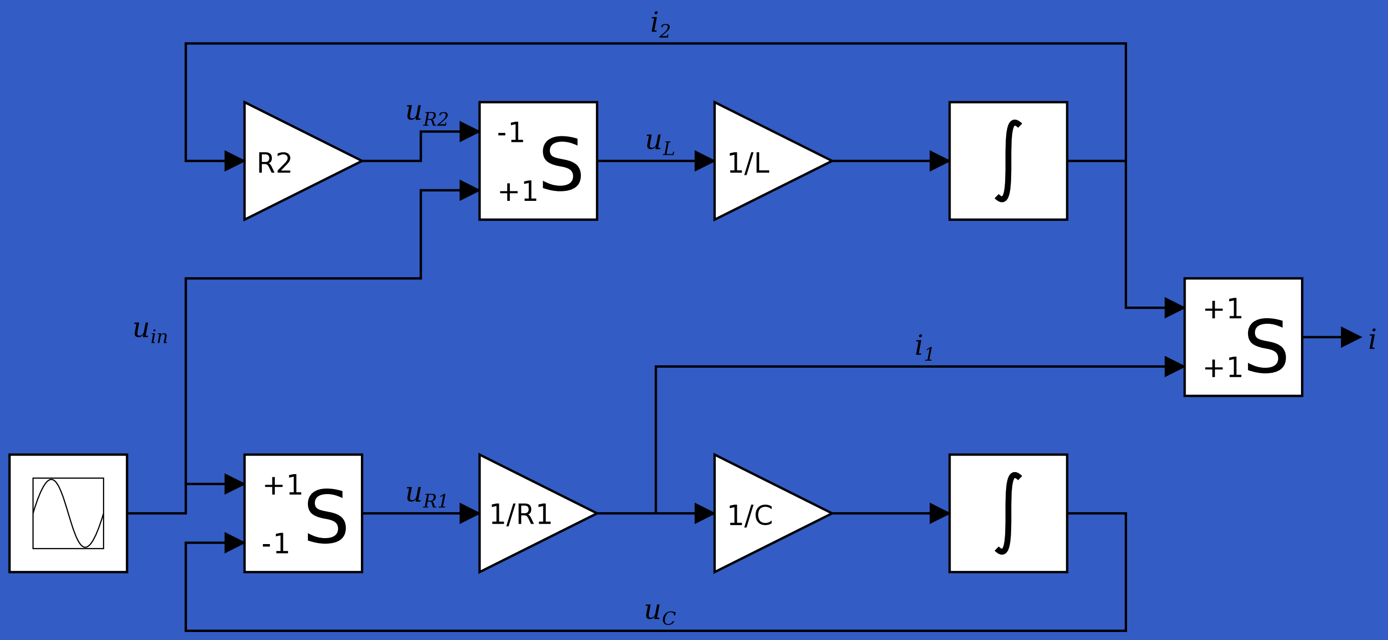
Yampa and arrows (2)

But systems can be complex:



Yampa and arrows (2)

But systems can be complex:



How many and what combinators do we need to be able to describe arbitrary systems?

Yampa and arrows (3)

John Hughes' ***arrow*** framework:

- Abstract data type interface for function-like types.

Yampa and arrows (3)

John Hughes' ***arrow*** framework:

- Abstract data type interface for function-like types.
- Particularly suitable for types representing process-like computations.

Yampa and arrows (3)

John Hughes' ***arrow*** framework:

- Abstract data type interface for function-like types.
- Particularly suitable for types representing process-like computations.
- Related to ***monads***, since arrows are (effectful) computations, but more general: any monad m induces an arrow, the Kleisli arrow, $\alpha \rightarrow m\ \beta$, but not vice versa.

Yampa and arrows (3)

John Hughes' ***arrow*** framework:

- Abstract data type interface for function-like types.
- Particularly suitable for types representing process-like computations.
- Related to ***monads***, since arrows are (effectful) computations, but more general: any monad m induces an arrow, the Kleisli arrow, $\alpha \rightarrow m\ \beta$, but not vice versa.
- Provides a minimal set of "wiring" combinators.

What is an arrow? (1)

- A ***type constructor*** a of arity two.

What is an arrow? (1)

- A ***type constructor*** a of arity two.
- Three operators:

What is an arrow? (1)

- A ***type constructor*** `a` of arity two.
- Three operators:
  - ***lifting***:

```
arr :: (b->c) -> a b c
```

What is an arrow? (1)

- A ***type constructor*** `a` of arity two.
- Three operators:
  - ***lifting***:
    `arr :: (b->c) -> a b c`
  - ***composition***:
    `(>>>) :: a b c -> a c d -> a b d`

What is an arrow? (1)

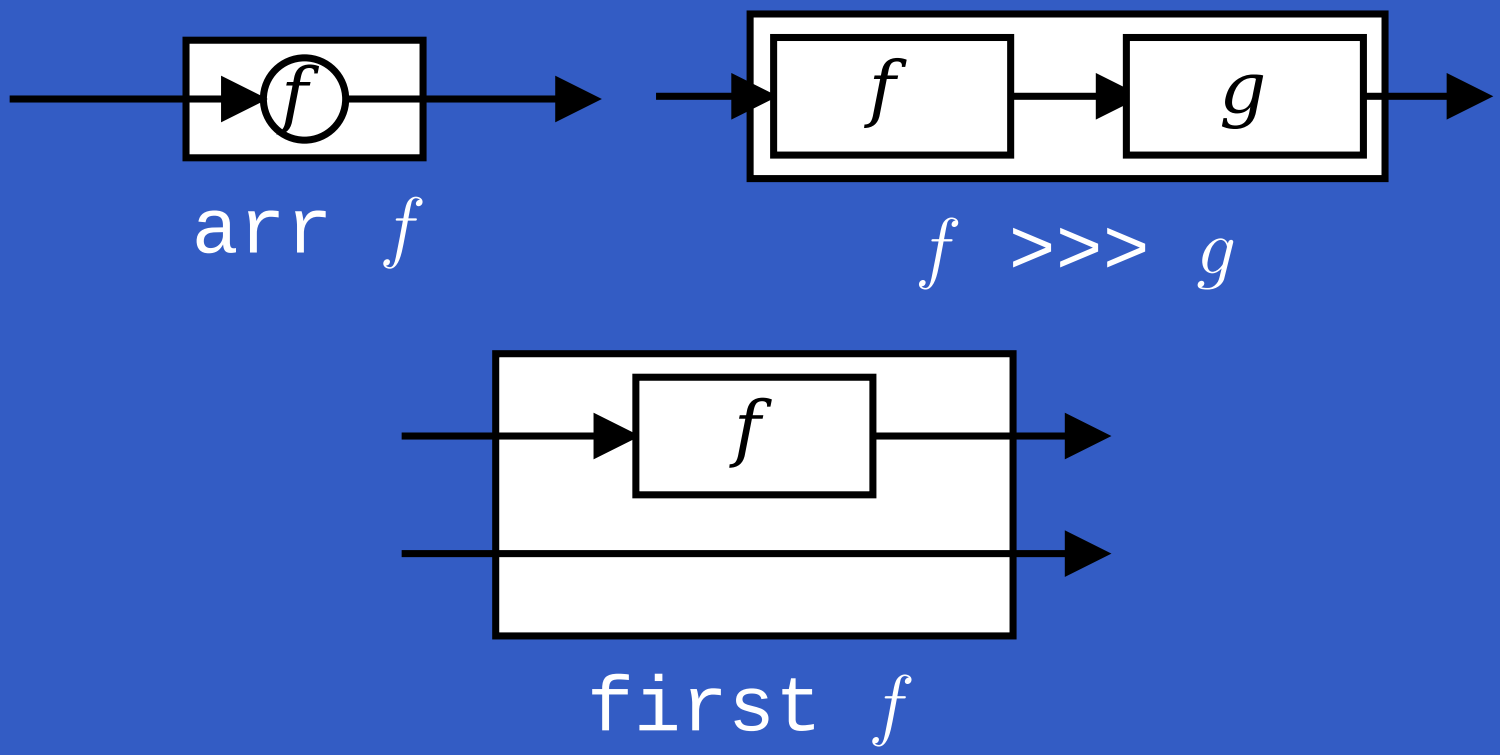
- A ***type constructor*** a of arity two.
- Three operators:
  - ***lifting***:
    `arr :: (b->c) -> a b c`
  - ***composition***:
    `(>>>) :: a b c -> a c d -> a b d`
  - ***widening***:
    `first :: a b c -> a (b,d) (c,d)`

What is an arrow? (1)

- A ***type constructor*** a of arity two.
- Three operators:
  - ***lifting***:
    `arr :: (b->c) -> a b c`
  - ***composition***:
    `(>>>) :: a b c -> a c d -> a b d`
  - ***widening***:
    `first :: a b c -> a (b,d) (c,d)`
- A set of ***algebraic laws*** that must hold.

What is an arrow? (2)

These diagrams convey the general idea:



The `Arrow` class

In Haskell, a ***type class*** is used to capture these ideas (except for the laws):

```
class Arrow a where
    arr   :: (b -> c) -> a b c
    (>>>) :: a b c -> a c d -> a b d
    first :: a b c -> a (b,d) (c,d)
```

Arrow laws

```
(f >>> g) >>> h  =  f >>> (g >>> h)
```

Arrow laws

```
(f >>> g) >>> h  =  f >>> (g >>> h)
    arr (g . f)  =  arr f >>> arr g
```

Arrow laws

```
(f >>> g) >>> h  =  f >>> (g >>> h)
    arr (g . f)  =  arr f >>> arr g
  arr id >>> f   =  f
```

Arrow laws

```
(f >>> g) >>> h  =  f >>> (g >>> h)
    arr (g . f)  =  arr f >>> arr g
   arr id >>> f  =  f
              f  =  f >>> arr id
```

Arrow laws

```
(f >>> g) >>> h  =  f >>> (g >>> h)
    arr (g . f)  =  arr f >>> arr g
   arr id >>> f  =  f
              f  =  f >>> arr id
  first (arr f)  =  arr (f × id)
```

Arrow laws

$$
\begin{aligned}
\texttt{(f >>> g) >>> h} &= \texttt{f >>> (g >>> h)} \\
\texttt{arr (g . f)} &= \texttt{arr f >>> arr g} \\
\texttt{arr id >>> f} &= \texttt{f} \\
\texttt{f} &= \texttt{f >>> arr id} \\
\texttt{first (arr f)} &= \texttt{arr (f} \times \texttt{id)} \\
\texttt{first (f >>> g)} &= \texttt{first f >>> first g}
\end{aligned}
$$

Arrow laws

$$
\begin{aligned}
\texttt{(f >>> g) >>> h} &= \texttt{f >>> (g >>> h)} \\
\texttt{arr (g . f)} &= \texttt{arr f >>> arr g} \\
\texttt{arr id >>> f} &= \texttt{f} \\
\texttt{f} &= \texttt{f >>> arr id} \\
\texttt{first (arr f)} &= \texttt{arr (f} \times \texttt{id)} \\
\texttt{first (f >>> g)} &= \texttt{first f >>> first g} \\
\texttt{first f >>> arr (id} \times \texttt{g)} &= \texttt{arr (id} \times \texttt{g) >>> first f}
\end{aligned}
$$

Arrow laws

$$
\begin{aligned}
\texttt{(f >>> g) >>> h} &= \texttt{f >>> (g >>> h)} \\
\texttt{arr (g . f)} &= \texttt{arr f >>> arr g} \\
\texttt{arr id >>> f} &= \texttt{f} \\
\texttt{f} &= \texttt{f >>> arr id} \\
\texttt{first (arr f)} &= \texttt{arr (f} \times \texttt{id)} \\
\texttt{first (f >>> g)} &= \texttt{first f >>> first g} \\
\texttt{first f >>> arr (id} \times \texttt{g)} &= \texttt{arr (id} \times \texttt{g) >>> first f} \\
\texttt{first f >>> arr fst} &= \texttt{arr fst >>> f}
\end{aligned}
$$

Arrow laws

$$
\begin{array}{rcl}
\texttt{(f >>> g) >>> h} & = & \texttt{f >>> (g >>> h)} \\
\texttt{arr (g . f)} & = & \texttt{arr f >>> arr g} \\
\texttt{arr id >>> f} & = & \texttt{f} \\
\texttt{f} & = & \texttt{f >>> arr id} \\
\texttt{first (arr f)} & = & \texttt{arr (f} \times \texttt{id)} \\
\texttt{first (f >>> g)} & = & \texttt{first f >>> first g} \\
\texttt{first f >>> arr (id} \times \texttt{g)} & = & \texttt{arr (id} \times \texttt{g) >>> first f} \\
\texttt{first f >>> arr fst} & = & \texttt{arr fst >>> f} \\
\texttt{first (first f) >>> arr assoc} & = & \texttt{arr assoc >>> first f}
\end{array}
$$

Functions are arrows (1)

Functions are a simple example of arrows. The arrow type constructor is just `(->)` in that case.

Exercise 1: Suggest suitable definitions of

- `arr`
- `(>>>)`
- `first`

for this case!

Functions are arrows (2)

Solution:

- `arr = id`

Functions are arrows (2)

Solution:

- `arr = id`

  To see this, recall

  ```
  id :: t -> t
  arr :: (b->c) -> a b c
  ```

Functions are arrows (2)

Solution:

- `arr = id`

  To see this, recall

  ```
  id :: t -> t
  arr :: (b->c) -> a b c
  ```

  Instantiate with

  $$\texttt{a} = \texttt{(->)}$$
  $$\texttt{t} = \texttt{b->c} = \texttt{(->) b c}$$

Functions are arrows (3)

- `f >>> g = \a -> g (f a)`

Functions are arrows (3)

- `f >>> g = \a -> g (f a)`   ***or***
- `f >>> g = g . f`

Functions are arrows (3)

- `f >>> g = \a -> g (f a)` ***or***
- `f >>> g = g . f` ***or even***
- `(>>>) = flip (.)`

Functions are arrows (3)

- `f >>> g = \a -> g (f a)` ***or***
- `f >>> g = g . f` ***or even***
- `(>>>) = flip (.)`
- `first f = \(b,d) -> (f b,d)`

Functions are arrows (4)

`Arrow` instance declaration for functions:

```
instance Arrow (->) where
    arr     = id
    (>>>)   = flip (.)
    first f = \(b,d) -> (f b,d)
```

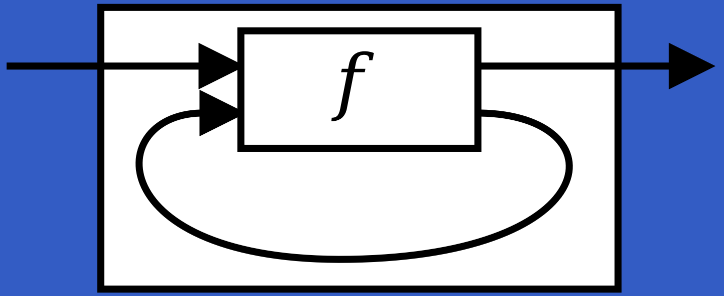
The arrow laws reformulated

Exploiting that functions are arrows, some of the laws can be formulated more neatly. E.g:

$$\mathtt{arr\ (f >>> g)} = \mathtt{arr\ f >>> arr\ g}$$
$$\mathtt{first\ (arr\ f)} = \mathtt{arr\ (first\ f)}$$

The `loop` combinator (1)

Another important operator is `loop`: a fixed-point operator used to express recursive arrows or ***feedback***:



`loop` f

The `loop` combinator (2)

Not all arrow instances support `loop`. It is thus a method of a separate class:

```
class Arrow a => ArrowLoop a where
    loop :: a (b, d) (c, d) -> a b c
```

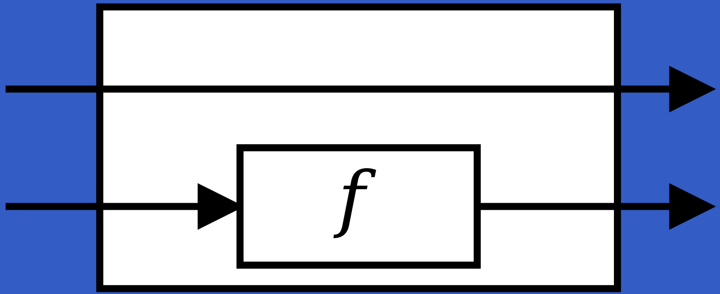
Remarkably, the four combinators `arr`, `>>>`, `first`, and `loop` are sufficient to express any conceivable wiring!

Some more arrow combinators (1)

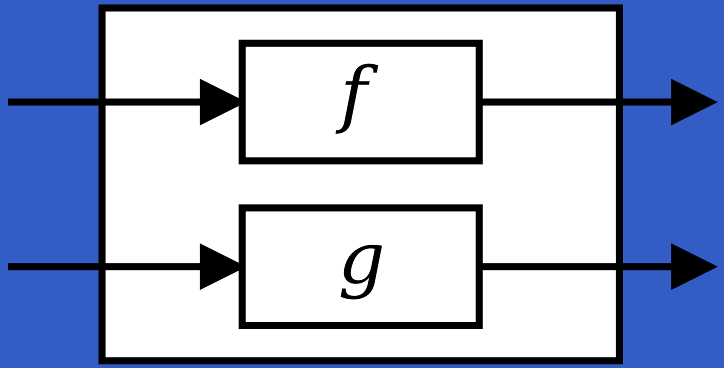
```
second :: Arrow a =>
    a b c -> a (d,b) (d,c)

(***) :: Arrow a =>
    a b c -> a d e -> a (b,d) (c,e)

(&&&) :: Arrow a =>
    a b c -> a b d -> a b (c,d)
```

Some more arrow combinators (2)

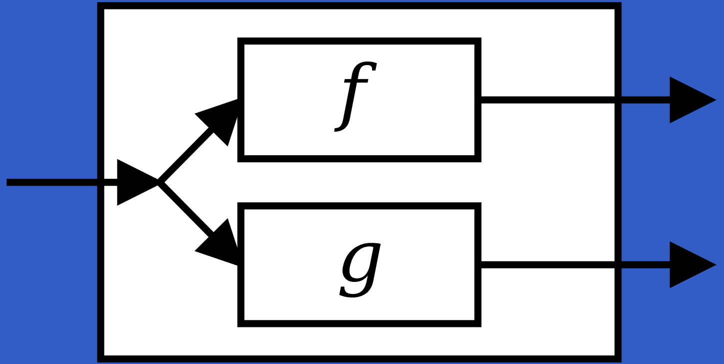
As diagrams:



second f



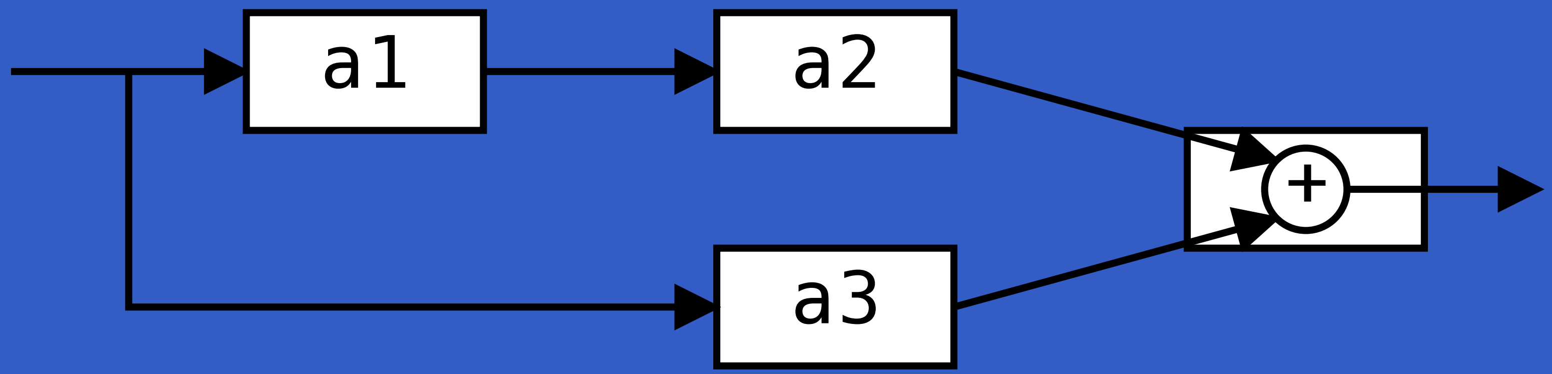
f *** g



f &&& g

Some more arrow combinators (3)

Exercise 2: Describe the following circuit using arrow combinators:

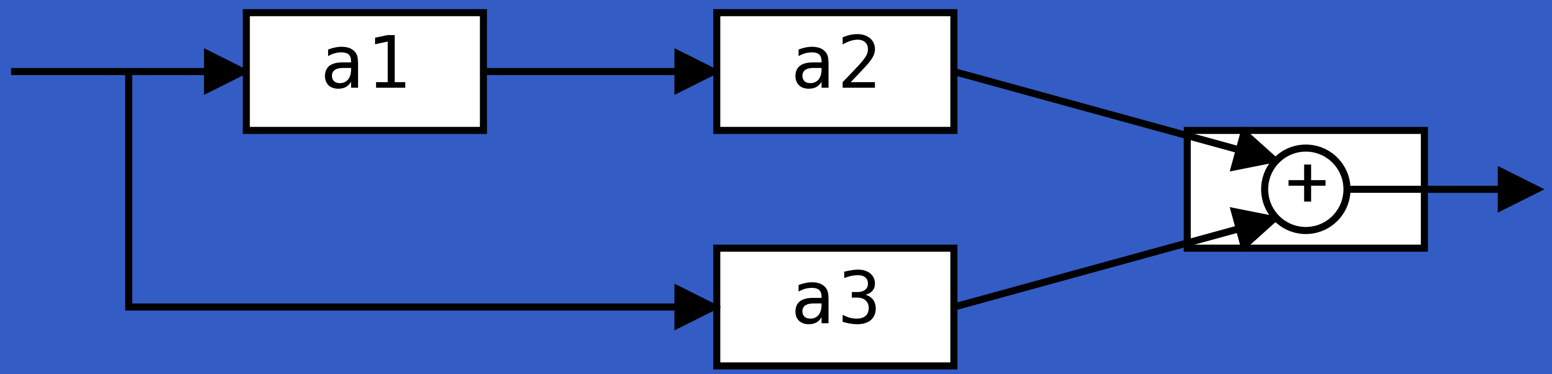


`a1, a2, a3 :: A Double Double`

Exercise 3: The combinators `second`, `(***)`, and `(&&&)` are not primitive, but defined in terms of `arr`, `(>>>)`, and `first`. Suggest suitable definitions!

Exercise 2: One solution

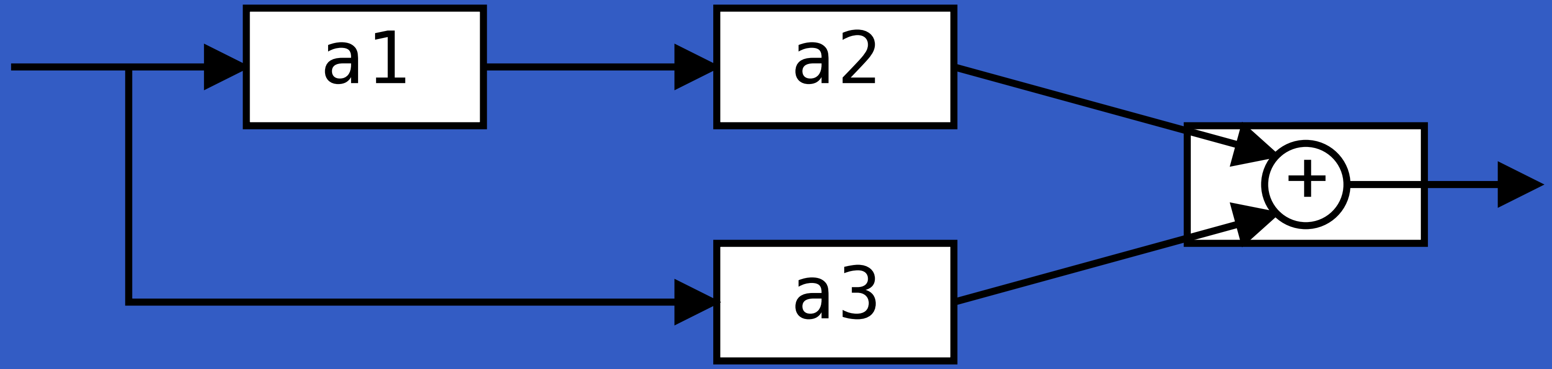
Exercise 2: Describe the following circuit using arrow combinators:



`a1, a2, a3 :: A Double Double`

Exercise 2: One solution

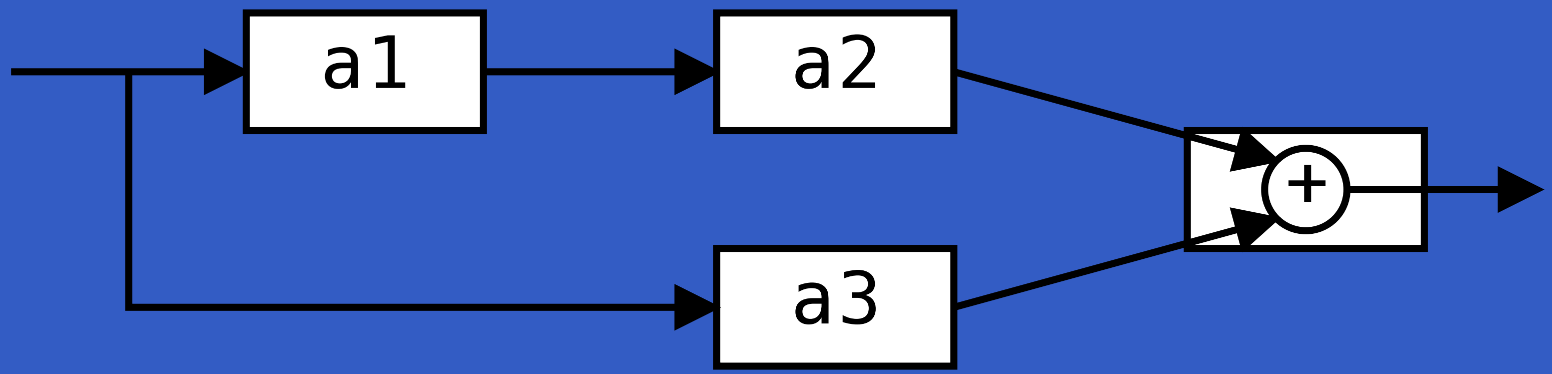
Exercise 2: Describe the following circuit using arrow combinators:




```
a1, a2, a3 :: A Double Double

circuit_v1 :: A Double Double
circuit_v1 = (a1 &&& arr id)
             >>> (a2 *** a3)
             >>> arr (uncurry (+))
```

Exercise 2: Another solution

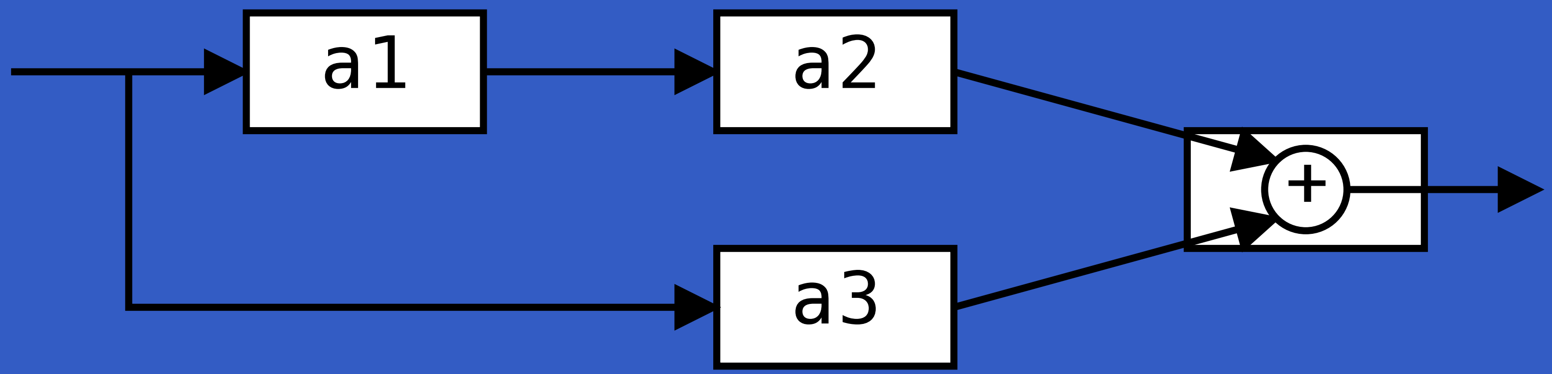
Exercise 2: Describe the following circuit:




```
a1, a2, a3 :: A Double Double
```

Exercise 2: Another solution

Exercise 2: Describe the following circuit:




```
a1, a2, a3 :: A Double Double

circuit_v2 :: A Double Double
circuit_v2 = arr (\x -> (x,x))
             >>> first a1
             >>> (a2 *** a3)
             >>> arr (uncurry (+))
```

Exercise 3: Solution

Exercise 3: Suggest definitions of `second`, `(***)`, and `(&&&)`.

Exercise 3: Solution

Exercise 3: Suggest definitions of `second`, `(***)`, and `(&&&)`.

```
second :: Arrow a => a b c -> a (d,b) (d,c)
second f = arr swap >>> first f >>> arr swap
swap (x,y) = (y,x)
```

Exercise 3: Solution

Exercise 3: Suggest definitions of `second`, `(***)`, and `(&&&)`.

```
second :: Arrow a => a b c -> a (d,b) (d,c)
second f = arr swap >>> first f >>> arr swap
swap (x,y) = (y,x)
```

```
(***) :: Arrow a =>
    a b c -> a d e -> a (b,d) (c,e)
f *** g = first f >>> second g
```

Exercise 3: Solution

Exercise 3: Suggest definitions of `second`, `(***)`, and `(&&&)`.

```
second :: Arrow a => a b c -> a (d,b) (d,c)
second f = arr swap >>> first f >>> arr swap
swap (x,y) = (y,x)
```

```
(***) :: Arrow a =>
    a b c -> a d e -> a (b,d) (c,e)
f *** g = first f >>> second g
```

```
(&&&) :: Arrow a => a b c -> a b d -> a b (c,d)
f &&& g = arr (\x->(x,x)) >>> (f *** g)
```

Note on the definition of (`***`) (1)

Are the following two definitions of (`***`) equivalent?

- `f *** g = first f >>> second g`
- `f *** g = second g >>> first f`

Note on the definition of `(***)` (1)

Are the following two definitions of `(***)` equivalent?

- `f *** g = first f >>> second g`
- `f *** g = second g >>> first f`

No, in general

$$\texttt{first}\ f \ \texttt{>>>}\ \texttt{second}\ g \neq \texttt{second}\ g\ \texttt{>>>}\ \texttt{first}\ f$$

since the ***order*** of the two possibly effectful computations f and g are different.

Note on the definition of (***) (2)

Similarly

$$(f *** g) >>> (h *** k) \neq (f >>> h) *** (g >>> k)$$

since the order of f and g differs.

Note on the definition of (***) (2)

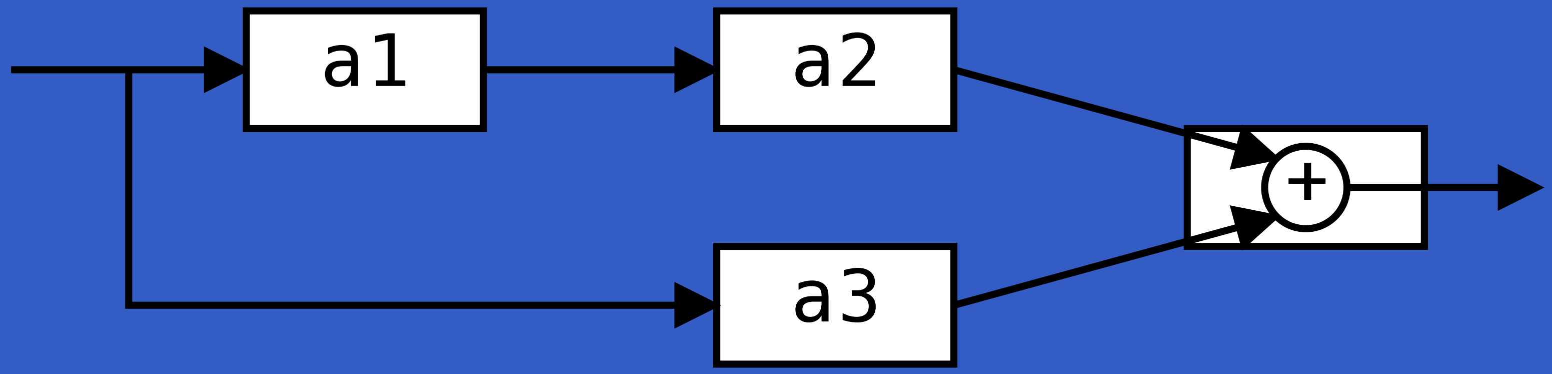
Similarly

$$(f *** g) >>> (h *** k) \neq (f >>> h) *** (g >>> k)$$

since the order of f and g differs.

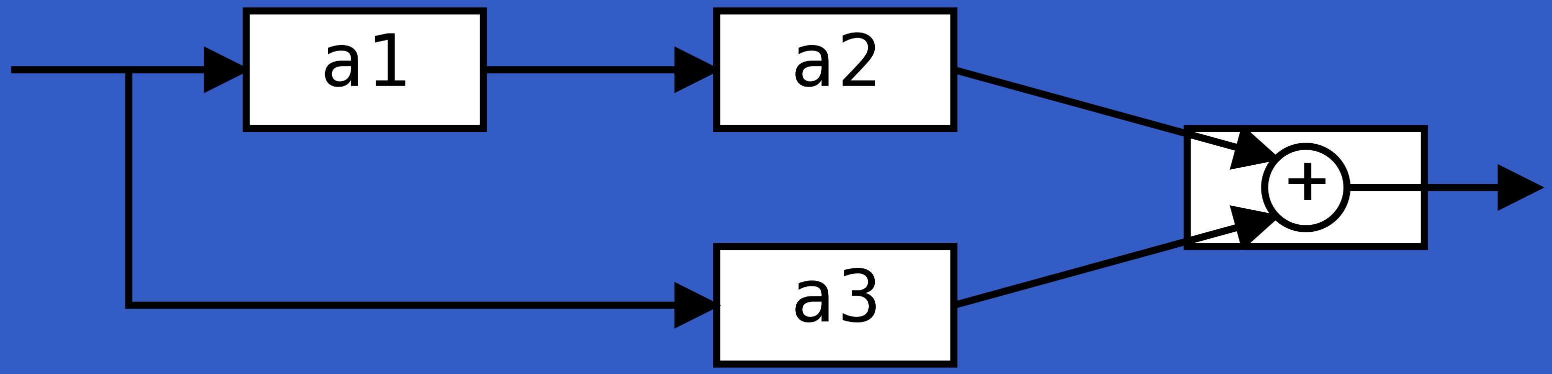
However, Yampa's signal functions have no effectful interaction: they are Causal Commutative Arrows (Liu, Cheng, Hudak 2009) ***Both*** considered identities actually hold.

Yet another attempt at exercise 2




```
circuit_v3 :: A Double Double
circuit_v3 = (a1 &&& a3)
             >>> first a2
             >>> arr (uncurry (+))
```

Yet another attempt at exercise 2




```
circuit_v3 :: A Double Double
circuit_v3 = (a1 &&& a3)
             >>> first a2
             >>> arr (uncurry (+))
```

Are `circuit_v1`, `circuit_v2`, and `circuit_v3` all equivalent?

Point-free vs. pointed programming

What we have seen thus far is an example of ***point-free*** programming: the values being manipulated are not given any names.

Point-free vs. pointed programming

What we have seen thus far is an example of ***point-free*** programming: the values being manipulated are not given any names.

This is often appropriate, especially for small definitions, and it facilitates equational reasoning as shown by Bird & Meertens (Bird 1990).

Point-free vs. pointed programming

What we have seen thus far is an example of ***point-free*** programming: the values being manipulated are not given any names.

This is often appropriate, especially for small definitions, and it facilitates equational reasoning as shown by Bird & Meertens (Bird 1990).

However, large programs are much better expressed in a ***pointed*** style, where names can be given to values being manipulated.

The arrow do notation (1)

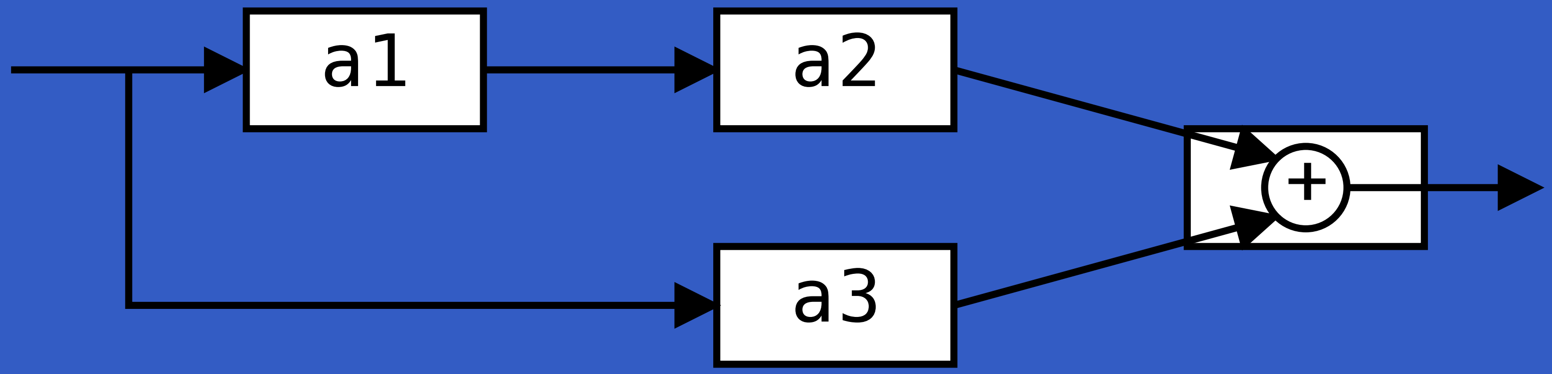
Ross Paterson's do-notation for arrows supports ***pointed*** arrow programming. Only ***syntactic sugar***.

$$
\begin{array}{l}
\texttt{proc}\ \mathit{pat}\ \texttt{->}\ \texttt{do}\ [\ \texttt{rec}\ ] \\
\qquad \mathit{pat}_1\ \texttt{<-}\ \mathit{sfexp}_1\ \texttt{-<}\ \mathit{exp}_1 \\
\qquad \mathit{pat}_2\ \texttt{<-}\ \mathit{sfexp}_2\ \texttt{-<}\ \mathit{exp}_2 \\
\qquad \texttt{...} \\
\qquad \mathit{pat}_n\ \texttt{<-}\ \mathit{sfexp}_n\ \texttt{-<}\ \mathit{exp}_n \\
\qquad \texttt{returnA}\ \texttt{-<}\ \mathit{exp}
\end{array}
$$

Also: $\texttt{let}\ \mathit{pat} = \mathit{exp} \quad \equiv \quad \mathit{pat}\ \texttt{<-}\ \texttt{arr id}\ \texttt{-<}\ \mathit{exp}$

The arrow do notation (2)

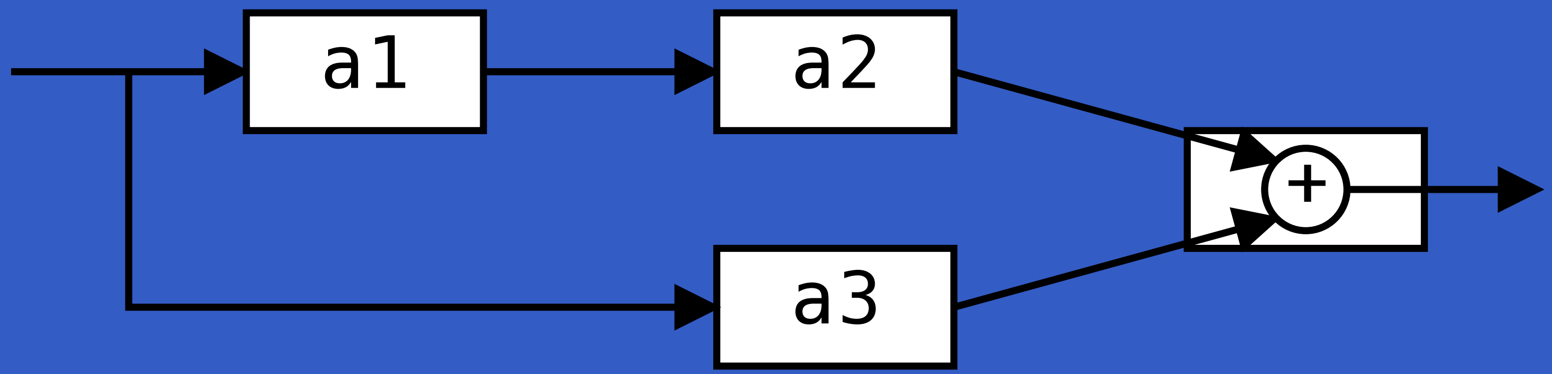
Let us redo exercise 3 using this notation:




```
circuit_v4 :: A Double Double
circuit_v4 = proc x -> do
    y1 <- a1 -< x
    y2 <- a2 -< y1
    y3 <- a3 -< x
    returnA -< y2 + y3
```

The arrow do notation (3)

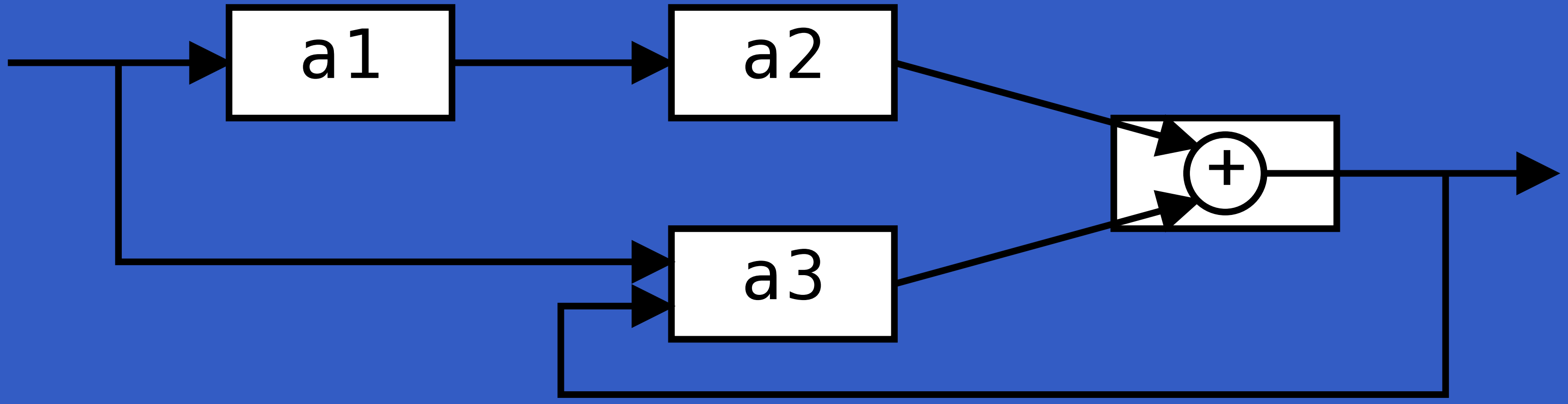
We can also mix and match:




```
circuit_v5 :: A Double Double
circuit_v5 = proc x -> do
    y2 <- a2 <<< a1 -< x
    y3 <- a3        -< x
    returnA -< y2 + y3
```

The arrow do notation (4)

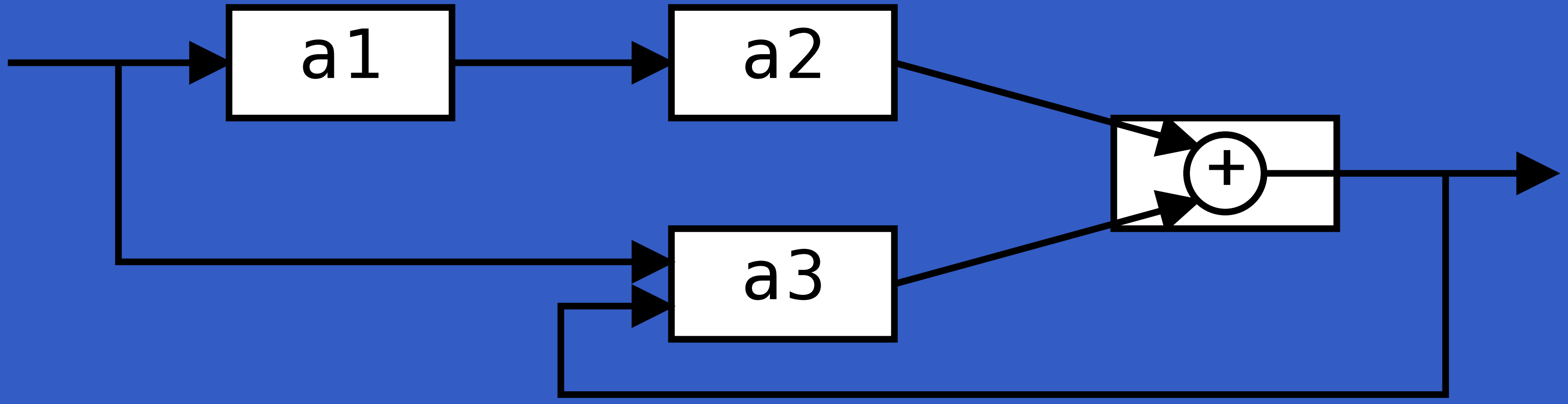
Exercise 4: Describe the following circuit using the arrow do-notation:




```
a1, a2 :: A Double Double
a3 :: A (Double,Double) Double
```

The arrow do notation (4)

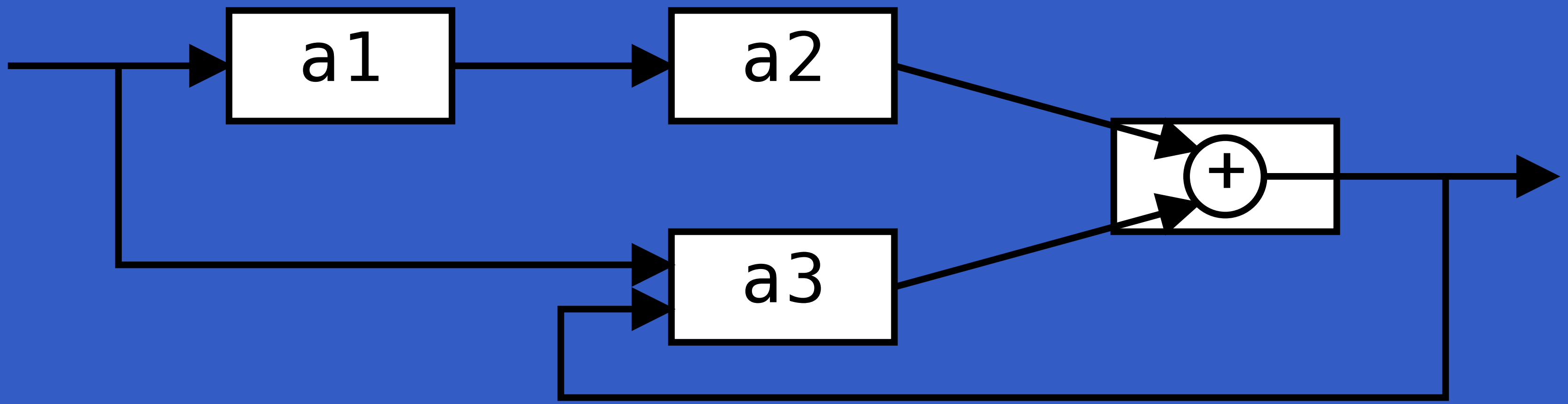
Exercise 4: Describe the following circuit using the arrow do-notation:




```
a1, a2 :: A Double Double
a3 :: A (Double,Double) Double
```

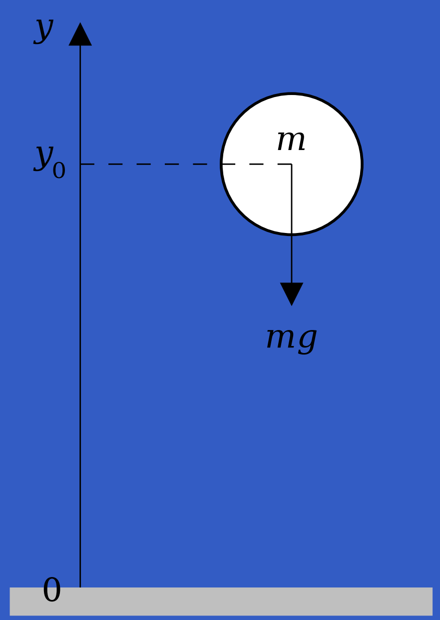
Exercise 5: As 4, but directly using only the arrow combinators.

Solution exercise 4




```
circuit = proc x -> do
    rec
        y1 <- a1 -< x
        y2 <- a2 -< y1
        y3 <- a3 -< (x, y)
        let y = y2 + y3
    returnA -< y
```

Some basic signal functions (1)

- ```
  identity :: SF a a
  identity = arr id
  ```
  ```

Some basic signal functions (1)

- ```
  identity :: SF a a
  identity = arr id
  ```
- ```
  constant :: b -> SF a b
  constant b = arr (const b)
  ```

Some basic signal functions (1)

- ```
  identity :: SF a a
  identity = arr id
  ```
- ```
  constant :: b -> SF a b
  constant b = arr (const b)
  ```
- `integral :: VectorSpace a s=>SF a a`
  It is defined through:

$$y(t) = \int_0^t x(\tau)\,\mathrm{d}\tau$$

Some basic signal functions (2)

- `iPre :: a -> SF a a`

Some basic signal functions (2)

- `iPre :: a -> SF a a`
- `(^<<) :: (b->c) -> SF a b -> SF a c`
  `f (^<<) sf = sf >>> arr f`

Some basic signal functions (2)

- `iPre :: a -> SF a a`
- `(^<<) :: (b->c) -> SF a b -> SF a c`
  `f (^<<) sf = sf >>> arr f`
- `time :: SF a Time`

Some basic signal functions (2)

- `iPre :: a -> SF a a`
- `(^<<) :: (b->c) -> SF a b -> SF a c`
  `f (^<<) sf = sf >>> arr f`
- `time :: SF a Time`

  Quick Exercise: Define time!

Some basic signal functions (2)

- `iPre :: a -> SF a a`
- `(^<<) :: (b->c) -> SF a b -> SF a c`
  `f (^<<) sf = sf >>> arr f`
- `time :: SF a Time`

  Quick Exercise: Define time!

  `time = constant 1.0 >>> integral`

Some basic signal functions (2)

- `iPre :: a -> SF a a`
- `(^<<) :: (b->c) -> SF a b -> SF a c`
  `f (^<<) sf = sf >>> arr f`
- `time :: SF a Time`

  Quick Exercise: Define time!

  ```
  time = constant 1.0 >>> integral
  ```

Note: there is ***no*** built-in notion of global time in Yampa: time is always ***local***, measured from when a signal function started.

A bouncing ball



$$
\begin{aligned}
y &= y_0 + \int v \, \mathrm{d}t \\
v &= v_0 + \int -9.81
\end{aligned}
$$

On impact:

$$v = -v(t-)$$

(fully elastic collision)

Modelling the bouncing ball: part 1

Free-falling ball:

```
type Pos = Double
type Vel = Double

fallingBall ::
    Pos -> Vel -> SF () (Pos, Vel)
fallingBall y0 v0 = proc () -> do
    v <- (v0 +) ^<< integral -< -9.81
    y <- (y0 +) ^<< integral -< v
    returnA -< (y, v)
```

Events

Conceptually, ***discrete-time*** signals are only defined at discrete points in time, often associated with the occurrence of some ***event***.

Events

Conceptually, ***discrete-time*** signals are only defined at discrete points in time, often associated with the occurrence of some ***event***.

Yampa models discrete-time signals by lifting the ***range*** of continuous-time signals:

```
data Event a = NoEvent | Event a
```

Events

Conceptually, ***discrete-time*** signals are only defined at discrete points in time, often associated with the occurrence of some ***event***.

Yampa models discrete-time signals by lifting the ***range*** of continuous-time signals:

```
data Event a = NoEvent | Event a
```

Discrete-time signal $=$ `Signal` $($`Event` $\alpha)$.

Events

Conceptually, ***discrete-time*** signals are only defined at discrete points in time, often associated with the occurrence of some ***event***.

Yampa models discrete-time signals by lifting the ***range*** of continuous-time signals:

```
data Event a = NoEvent | Event a
```

Discrete-time signal $= \texttt{Signal}\,(\texttt{Event}\,\alpha)$.

Associating information with an event occurrence:

```
tag :: Event a -> b -> Event b
```

Some basic event sources

- `never :: SF a (Event b)`
- `now :: b -> SF a (Event b)`
- `after :: Time -> b -> SF a (Event b)`
- ```
  repeatedly ::
      Time -> b -> SF a (Event b)
  ```
- `edge :: SF Bool (Event ())`
  ```

Stateful event suppression

- `notYet :: SF (Event a) (Event a)`
- `once :: SF (Event a) (Event a)`

Modelling the bouncing ball: part 2

Detecting when the ball goes through the floor:

```
fallingBall' ::
    Pos -> Vel
    -> SF () ((Pos,Vel), Event (Pos,Vel))
fallingBall' y0 v0 = proc () -> do
    yv@(y, _) <- fallingBall y0 v0 -< ()
    hit       <- edge              -< y <= 0
    returnA -< (yv, hit `tag` yv)
```

Switching

Q: How and when do signal functions “start”?

Switching

Q: How and when do signal functions “start”?

A:
- ***Switchers*** “apply” a signal functions to its input signal at some point in time.

Switching

Q: How and when do signal functions “start”?

A:
- ***Switchers*** “apply” a signal functions to its input signal at some point in time.
- This creates a “running” signal function ***instance***.

Switching

Q: How and when do signal functions "start"?

A:
- ***Switchers*** "apply" a signal functions to its input signal at some point in time.
- This creates a "running" signal function ***instance***.
- The new signal function instance often replaces the previously running instance.

Switching

Q: How and when do signal functions “start”?

A:
- ***Switchers*** “apply” a signal functions to its input signal at some point in time.
- This creates a “running” signal function ***instance***.
- The new signal function instance often replaces the previously running instance.

Switchers thus allow systems with ***varying structure*** to be described.

The basic switch (1)

Idea:

- Allows one signal function to be replaced by another.
- Switching takes place on the first occurrence of the switching event source.

```
switch ::
    SF a (b, Event c)
   -> (c -> SF a b)
   -> SF a b
```

The basic switch (1)

Idea:

- Allows one signal function to be replaced by another.
- Switching takes place on the first occurrence of the switching event source.

```
switch ::
    SF a (b, Event c)
    -> (c -> SF a b)
    -> SF a b
```

Initial SF with event source

The basic switch (1)

Idea:

- Allows one signal function to be replaced by another.
- Switching takes place on the first occurrence of the switching event source.

```
switch ::
    SF a (b, Event c)
   -> (c -> SF a b)
   -> SF a b
```

Function yielding SF to switch into

The basic switch (2)

Exercise 6: Define an event counter `countFrom`

```
countFrom ::
    Int -> SF (Event a) Int
```

using

```
switch   :: SF a (b, Event c)
            -> (c -> SF a b)
            -> SF a b
constant :: b -> SF a b
notYet   :: SF (Event a) (Event a)
```

and any other basic combinators you might need.

Solution exercise 6

```
countFrom :: Int -> SF (Event a) Int
countFrom n =
  switch
    (constant n &&& identity
    (const (notYet >>> countFrom (n+1)))
```

Solution exercise 6

Another version that ignores any event at time 0 also from the very start:

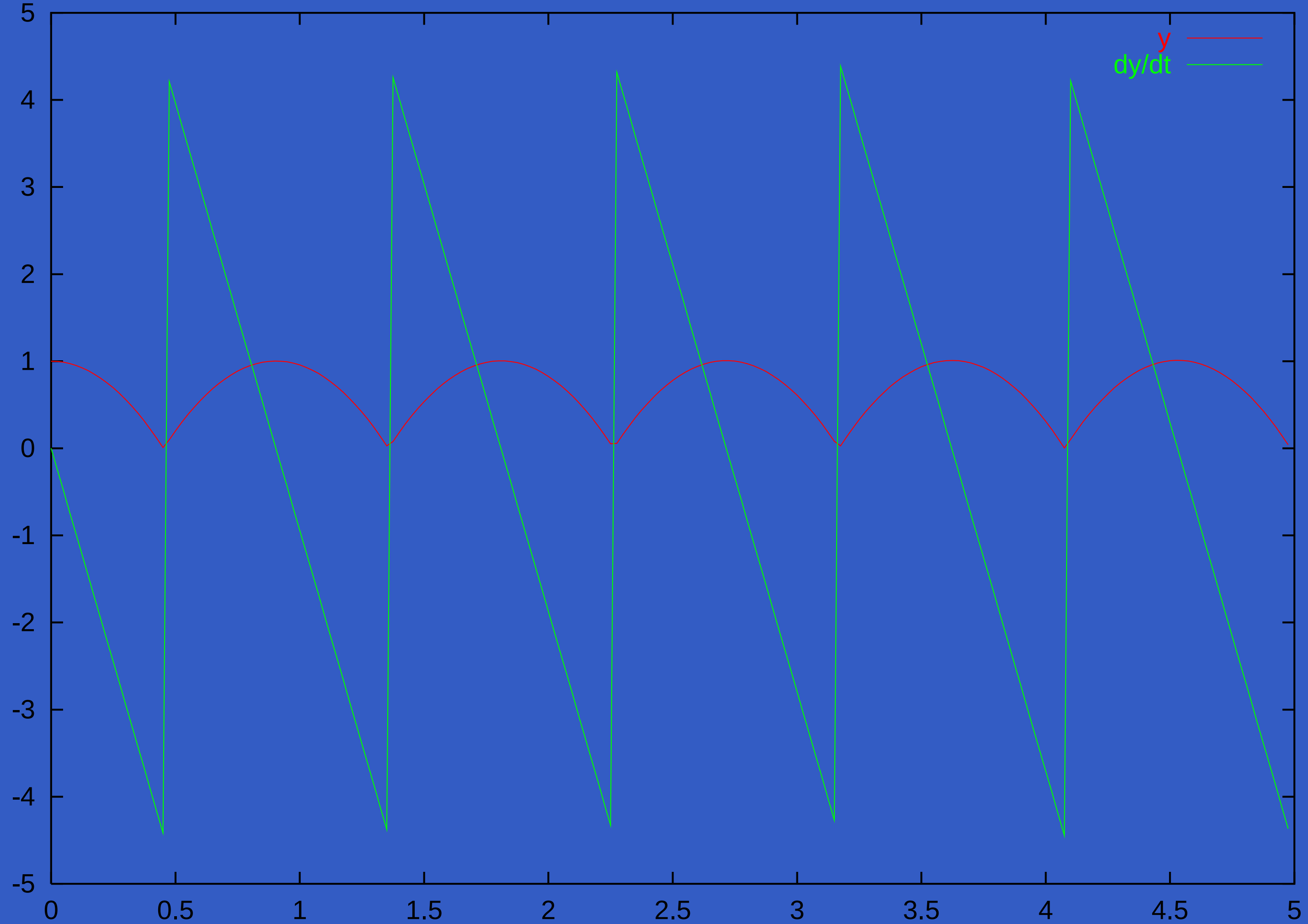
```
countFrom :: Int -> SF (Event a) Int
countFrom n =
  switch
    (constant n &&& notYet)
    (const (countFrom (n+1)))
```

Modelling the bouncing ball: part 3

Making the ball bounce:

```
bouncingBall :: Pos -> SF () (Pos, Vel)
bouncingBall y0 = bbAux y0 0.0
  where
    bbAux y0 v0 =
      switch (fallingBall’ y0 v0) $ \(y,v) ->
      bbAux y (-v)
```

Simulation of bouncing ball



Modelling using impulses (1)

From a modelling perspective, using a device like `switch` to model the interaction between the ball and the floor may seem rather unnatural.

Modelling using impulses (1)

From a modelling perspective, using a device like `switch` to model the interaction between the ball and the floor may seem rather unnatural.

A more appropriate account of what is going on is that an ***impulsive*** force is acting on the ball for a short time.

Modelling using impulses (1)

From a modelling perspective, using a device like `switch` to model the interaction between the ball and the floor may seem rather unnatural.

A more appropriate account of what is going on is that an ***impulsive*** force is acting on the ball for a short time.

This can be abstracted into ***Dirac Impulses***: impulses that act instantaneously. See

> Henrik Nilsson. Functional Automatic Differentiation with Dirac Impulses. In *Proceedings of ICFP 2003*.

Modelling using impulses (2)

However, Yampa does provide a derived version of `integral` capturing the basic idea:

```
impulseIntegral ::
    VectorSpace a k =>
        SF (a, Event a) a
```

The decoupled switch

```
dSwitch ::
        SF a (b, Event c)
        -> (c -> SF a b)
        -> SF a b
```

- Output at the point of switch is taken from the old subordinate signal function, ***not*** the new residual signal function.
- This means the ***output*** at the current point in time is ***independent*** of whether or not the ***switching event*** occurs at that point in time. Hence decoupled.

The recurring switch

```
rSwitch, drSwitch ::
    SF a b -> SF (a,Event (SF a b)) b
```

- Switching events received on the signal function input, carrying signal function to switch into.
- Switching occurs whenever an event occurs, not just once.
- Can be defined in terms of `switch`/`dSwitch`.

Reading (1)

- John Hughes. Generalising monads to arrows. *Science of Computer Programming*, 37:67–111, May 2000
- John Hughes. Programming with arrows. In *Advanced Functional Programming*, 2004. Springer Verlag.
- Ross Paterson. A New Notation for Arrows. In *Proceedings of the 2001 ACM SIGPLAN International Conference on Functional Programming*, pp. 229–240, Firenze, Italy, 2001.

Reading (2)

- Henrik Nilsson, Antony Courtney, and John Peterson. Functional reactive programming, continued. In *Proceedings of the 2002 Haskell Workshop*, pp. 51–64, October 2002.
- Paul Hudak, Antony Courtney, Henrik Nilsson, and John Peterson. Arrows, robots, and functional reactive programming. In *Advanced Functional Programming*, 2002. LNCS 2638, pp. 159–187.

Reading (3)

- Hai Liu, Eric Cheng and Paul Hudak. Causal Commutative Arrows and Their Optimization. In *The 14th ACM SIGPLAN International Conference on Functional Programming (ICFP 2009)*, Edinburgh, Scotland, September, 2009
- Richard S. Bird. A calculus of functions for program derivation. In *Research Topics in Functional Programming*, Addison-Wesley, 1990.